

CS422 – Formal Methods in System Design: A Monitor Synthesis Algorithm for Past LTL

Grigore Roşu

Department of Computer Science,
University of Illinois at Urbana-Champaign, USA.
`grosu@cs.uiuc.edu`

A monitor synthesis algorithm from linear temporal logic (LTL) safety formulae of the form $\Box\varphi$ where φ is a past time LTL formula was presented in [3]. The generated monitors implemented the recursive semantics of past-time LTL using a dynamic programming technique, and needed $O(|\varphi|)$ time to process each new event and $O(|\varphi|)$ total space. Some compiler-like optimizations of the generated monitors were also proposed in [3], which would further reduce the required space. It is not clear how much the required space could be reduced by applying those optimizations.

We here show how to generate using a divide-and-conquer technique directly monitors that need $O(k)$ space and still $O(|\varphi|)$ time, where k is the number of temporal operators in φ .

1 The Monitor Synthesis Algorithm

For simplicity, we assume only two past operators, namely \circ (previously) and \mathcal{S} (since). Let us first note that one cannot asymptotically reduce the space requirements below $\Omega(k)$, where k is the number of temporal operators appearing in the formula to monitor φ . Indeed, one can take $\varphi = (\#_1 \rightarrow t_1) \wedge \cdots \wedge (\#_k \rightarrow t_k)$, where for each $1 \leq i \leq k$, $\#_i$ is some event and t_i is some temporal formula containing precisely one past temporal operator, i.e., a \circ or a \mathcal{S} . Any monitor for φ must directly or indirectly store the status of each t_i at every event, to be able to react accordingly in case the next event is some $\#_i$. Assuming that the events $\#_i$ are distinct and that the formulae t_i are unrelated, then the monitor needs to distinguish among 2^k possible states, so it needs $\Omega(k)$ space.

In what follows, we assume the usual recursive semantics of LTL, also presented below, restricted to safety formulae of the form $\Box\varphi$, where φ is a past-time LTL. We adopt the simplifying assumption that the empty trace invalidates any atomic proposition and any past temporal operator; as argued in [3], this may not always be the best choice, but other semantic variations regarding the empty trace present no difficulties for monitoring.

Definition 1. (adapted from [4]) *LTL formulae of the form $\Box\varphi$ (read “always φ ”), where φ is a past-time LTL formula, are called LTL safety formulae; we may call them just safety formulae when LTL is understood from the context. An infinite trace $u \in \Sigma^\omega$ satisfies $\Box\varphi$, written $u \models \Box\varphi$, iff each $w \in \text{prefixes}(u)$*

satisfies the past-time LTL formula φ , written also $w \models \varphi$ and defined inductively as follows:

$$\begin{array}{ll}
w \models \text{true} & \text{is always true,} \\
ws \models a & \text{iff } a(s) \text{ holds,} \\
w \models \neg\varphi & \text{iff } w \not\models \varphi, \\
w \models \varphi_1 \wedge \varphi_2 & \text{iff } w \models \varphi_1 \text{ and } w \models \varphi_2, \\
ws \models \odot\varphi & \text{iff } w \models \varphi, \\
ws \models \varphi_1 \mathcal{S} \varphi_2 & \text{iff } ws \models \varphi_2 \text{ or } ws \models F_\varphi \text{ and } w \models \varphi_1 \mathcal{S} \varphi_2 \\
\epsilon \models \varphi & \text{is false otherwise}
\end{array}$$

Given safety formula $\Box\varphi$, we let $\mathcal{L}(\Box\varphi) \subseteq \Sigma^\omega$ be the set $\{u \in \Sigma^\omega \mid u \models \Box\varphi\}$.

Let us next investigate the problem of monitoring safety properties P expressed as languages of safety formulae, that is, $P = \mathcal{L}(\Box\varphi)$ for some past-time LTL formula φ . Because of the recursive nature of the satisfaction relation, a first important observation is that the generated monitor only needs to store information regarding the status of temporal operators from the previous state. More precisely, the monitor needs one bit per temporal operator, keeping the satisfaction status of the subformula corresponding to that temporal operator; when a new state is received, the satisfaction status of the subformula is recalculated according to the recursive semantics above and then the bit is updated. The order in which the temporal operators are processed when a new state is received is important: the nested operators must be processed first.

We next present the actual monitor synthesis algorithm at a high-level. We refrain from giving detailed pseudocode as we did in [3], because different applications may choose different implementation paradigms. For example, we are currently using rewriting techniques to implement the monitor synthesis algorithms in MOP [1]; Section 1.1 shows our complete Maude rewriting implementation of the subsequent monitor synthesis algorithm.

- Step 1** Let $\varphi_1, \dots, \varphi_k$ be the k subformulae of φ corresponding to temporal operators, such that, if φ_i is a subformula of φ_j , then $i < j$; this can be easily achieved by a DFS traversal of φ .
- Step 2** Let $bit[1..k]$ be a vector of k bits initialized to 0 (or false); $bit[i]$ will store information related to φ_i from the previous state:
 - if $\varphi_i = \odot\psi$ then $bit[i]$ says if ψ was satisfied at the previous state;
 - if $\varphi_i = \psi \mathcal{S} \psi'$ then $bit[i]$ says if φ_i was satisfied at the previous step.
- Step 3** Let $bit'[1..k]$ be another vector of k bits; this will be used to store temporary results, which will be moved eventually into the vector $bit[1..k]$.
- Step 4** Generate a loop that executes whenever a new state s is available; the body of the loop executes the following code:
 - Step 4.1** For each i from 1 to k execute a bit assignment as follows, where for a subformula ψ of φ , $\overline{\psi}$ is the boolean expression replacing in ψ each non-nested temporal subformula φ_j by $bit[j]$ if φ_j is a “previously” formula or by $bit'[j]$ if φ_j is a “since” formula, and each remaining atomic proposition a by its satisfaction in the current state, $a(s)$:

- if $\varphi_i = \odot\psi$ then generate the assignment $bit'[i] := \overline{\psi}$
- if $\varphi_i = \psi \mathcal{S} \psi'$ then generate the assignment $bit'[i] := \overline{\psi'} \vee \overline{\psi} \wedge bit[i]$

Step 4.2 Generate the conditional: if $\overline{\varphi}$ is false then error (formula violated)

Step 4.3 Generate code to move the contents of $bit'[1..k]$ into $bit[1..k]$.

Note that the generated monitors are well-defined, because each time a $\overline{\psi}$ boolean expression is generated, all the bits in $bit'[1..k]$ that are needed are already calculated. One can also perform boolean simplifications when calculating $\overline{\psi}$ to reduce runtime overhead even further. For example, in our implementation that also generated the code below (see Section 1.1), we used the simplification $\neg\neg\psi = \psi$. To illustrate the monitor generation algorithm above, let us consider the past time LTL formula: $\varphi = \neg(a \wedge \neg(\odot b \wedge (c \mathcal{S} (d \wedge (\neg e \mathcal{S} f))))$. Step 1 produces the following enumeration of φ 's subformulae: $\varphi_1 = \odot b$, $\varphi_2 = \neg e \mathcal{S} f$, and $\varphi_3 = c \mathcal{S} (d \wedge (\neg e \mathcal{S} f))$. The other steps eventually generate the code:

```

bit[1..3] := false;      // three global bits
foreach new state s do {
    // first update the bits in a consistent order
    bit'[1] := b(s);
    bit'[2] := f(s) ∨ (¬e(s) ∧ bit[2]);
    bit'[3] := d(s) ∧ bit'[2] ∨ (c(s) ∧ bit[3]);
    // then check whether the formula is violated
    if a(s) ∧ ¬(bit[1] ∧ bit'[3]) then Error;
    // finally, update the state of the monitor
    bit[1..3] := bit'[1..3]
}

```

It is easy to see that for any past LTL formula φ of k temporal operators, the state of the generated monitor is encoded on k bits, namely the vector $bit[1..k]$. The runtime of the generated monitor is still $O(|\varphi|)$, because each temporal operator in φ results in an assignment and a read operation in the monitor, while each boolean operator in φ is “executed” by the monitor.

1.1 A Maude Implementation of the Monitor Synthesizer

We here show a term rewriting implementation of the algorithm above, using the Maude system [2]. Implementations in other languages are obviously also possible; however, rewriting proved to be an elegant means to generate monitors from logical formulae in several other contexts, and so seems to be here. In what follows we show the complete Maude code that takes as input a formula, parses it, generates the monitor, and then pretty prints it. We use the **K** technique here [5], which is a rewriting-based language and/or logic definitional technique; to use **K**, one needs to first upload the generic, i.e., application-independent, module discussed at the end of this section.

Atomic Predicates We start by defining the atomic state predicates that one can use in formulae. These can be either identifiers (of the form 'a', 'abc', 'a123, etc.; these are provided by the Maude builtin module QID):

```
fmod PREDICATE is
  --- atomic predicates can be quoted identifiers
  protecting QID .
  sort Predicate .
  subsort Qid < Predicate .
endfm
```

Syntax of Formulae Let us next define the syntax of formulae. We here use Maude's mixfix notation for defining syntax as algebraic operators, where underscores stay for arguments. Also, note that operators are assigned precedences (declared as operator attributes), to relive the user from writing parentheses (the lower the precedence the tighter the binding):

```
fmod SYNTAX is
  protecting PREDICATE .
  sort Formula .
  subsort Predicate < Formula .
  op !_ : Formula -> Formula [prec 20] .
  op _/\_ : Formula Formula -> Formula [prec 23] .
  op !_ : Formula -> Formula [prec 21] .
  op !_ : Formula Formula -> Formula [prec 22] .
endfm
```

Target Language We are done with the input language. Let us now define the output language. We need a very simple language for implementing the generated monitors, namely one with limited assignment, conditional and looping. The generated code, as well as the target language, play no role in this paper; one is expected to change the language below to one's desired target language (Java, C, C#, assembler, etc.). Our chosen language below has bits, expressions, statements and code. Bits are also expressions; code is a list of statements composed sequentially using ";" or just concatenation. The syntax below is also making use of precedence attributes. The `format` attributes are useful solely for pretty-printing reasons (see Maude's manual [2] for details on formatting):

```
fmod CODE is
  --- syntax for the generated code
  protecting PREDICATE + INT + STRING .
  sorts Bit Exp Statement Code .
  subsorts Bit < Exp .
  subsort Statement < Code .
  ops (bit[_]) (bit'[_]) : Nat -> Bit .
  ops (bit[1 .. _]) (bit'[1 .. _]) : Int -> Bit .
  op _(s) : Predicate -> Exp [prec 0] .
  ops true false : -> Exp .
```

```

op !_ : Exp -> Exp [prec 20] .
op _/\_ : Exp Exp -> Exp [prec 23] .
op _\/_ : Exp Exp -> Exp [prec 24] .
op _:=_ : Exp Exp -> Statement [prec 27 format(ni d d d)] .
op if_then_ : Exp Statement -> Statement
      [format(ni d d ++ --) prec 30] .
op foreach new state s do _ : Code -> Statement
      [format(n d d d s++ --n)] .
op Error : -> Statement [format(ni d)] .
op //_ : String -> Statement [format(ni d d)] .
op nil : -> Code .
op _;_ : Code Code -> Code [assoc id: nil prec 40] .
op __ : Code Code -> Code [assoc id: nil prec 40] .
op {_} : Code -> Statement [format(d d --ni ++)] .
--- code simplification rules
var B : Exp .
eq !! B = B .
endfm

```

The following module defines the actual monitor synthesis algorithm. We use the K definitional technique here, because it yields a very compact implementation. K is centered on the basic intuition of *computation*; computations are encoded as first-order data-structures that “evolve”, via rewriting, to *results*. Computations are sequentialized using the list constructor “ $_{-} \rightarrow _{-}$ ”; thus, if K and K' are computations, then $K \rightarrow K'$ is the computation consisting of K followed by K' . Computations may eventually yield results; for example, $K \rightarrow K'$ may rewrite (in context) to $R \rightarrow K'$, meaning that R is the result that K reduces to. An important feature of K is that one can schedule lists of tasks for reduction; for example, $[K1, K2, K3] \rightarrow K$ may eventually reduce to $[R1, R2, R3] \rightarrow K$, where $R1$, $R2$, and $R3$ are the results that $K1$, $K2$, and $K3$ reduce to, in this order. To use K , one needs to import the module K discussed at the end of this section. The equations of the module K (three in total) are all about reducing a list of computations to a list of results, supposing that one knows how to reduce one computation to one result.

K is a definitional framework that is generic in computations and results. More precisely, it provides sorts `KComputation` and `KResult`, and expects its user to define the desired computations and results, as well as rules to reduce a computation to a result. Computations typically can be reduced to results only in context; to facilitate this, K provides a sort `KConfiguration`, which is also supposed to be populated accordingly. The sort `KConfiguration` is a multi-set sort over a sort `KConfigurationItem`, where the multi-set constructor is just concatenation; also, the sort `KComputation` is a list sort over `KComputationItem`, where the list constructor is $_{-} \rightarrow _{-}$. To make use of K , one needs to first define constructors for the sorts `KConfigurationItem`, `KComputationItem` and `KResult`, and then to define how each computation item reduces to a result.

In our case, the computations are the formulae or subformulae that still need to be processed, and the results are the corresponding boolean expressions that

need to be checked in the current (generated code) context to see whether the formula has been violated or not. We define the following additional constructors: we add four constructors for configurations, namely “**k**” that wraps the current computation, “**code**” that wraps the current generated code, and “**nextBit**” that wraps the next available bit; we add one main constructor for computations, “**form**”, that wraps a formula, and one constant computation item per operator in the input language (the later is needed to know how to combine back the results of the corresponding subexpressions; finally, we add one constructor for results, “**exp**”, that wraps a boolean expression.

The formula is processed in a depth-first-order, following a divide-and-conquer philosophy. Each subformula is decomposed into a list of computation subtasks consisting of its subformulae, then the corresponding results are composed back into a result corresponding to the original subformula. Recall that equations/rules apply wherever they match, not only at the top. Let us only discuss the two equations defining the “since” (**S**), the last two in the module below. The first one is straightforward: it decomposes the task of processing **F1 S F2** to the subtasks of processing **F1** and **F2**; the computation item **S** is placed in the computation structure to prepare the terrain for the next equation. The next equation applies after **F1** and **F2** have been processed, say to expressions **B1** and **B2**, respectively; if **C** is the code generated so far and if **I+1** is the next bit available, then the boolean expression corresponding to the current since formula is indeed **bit' (I+1)**, provided that one adds the corresponding code capturing the recursive semantics of since to the generated code.

```
fmod MONITOR-GENERATION is
  protecting K + SYNTAX + CODE .
  op k : KComputation -> KConfigurationItem .
  op code : Code -> KConfigurationItem .
  op nextBit : Nat -> KConfigurationItem .
  op process : Formula -> KConfiguration .
  op form : Formula -> KComputationItem .
  op exp : Exp -> KResult .
  ops ! /\ 0 S : -> KComputationItem .
  var P : Predicate . vars F F1 F2 : Formula . var C : Code .
  var I : Nat . vars B B1 B2 : Exp . var K : KComputation .
  eq process(F) = k(form(F)) code(nil) nextBit(0) .
  eq k(form(P) -> K) = k(exp(P(s)) -> K) .
  eq form(! F) = form(F) -> ! .
  eq exp(B) -> ! = exp(! B) .
  eq form(F1 /\ F2) = [form(F1),form(F2)] -> /\ .
  eq [exp(B1),exp(B2)] -> /\ = exp(B1 /\ B2) .
  eq form(0 F) = form(F) -> 0 .
  eq k(exp(B) -> 0 -> K) code(C) nextBit(I)
    = k(exp(bit[I + 1]) -> K) code(C ; bit'[I + 1] := B)
      nextBit(I + 1) .
  eq form(F1 S F2) = [form(F1), form(F2)] -> S .
  eq k([exp(B1),exp(B2)] -> S -> K) code(C) nextBit(I)
    = k(exp(bit'[I + 1]) -> K)
```

```

        code(C ; bit'[I + 1] := B2 /\ B1 /\ bit[I + 1]) nextBit(I + 1) .
    endfm

```

Putting It All together The following module plugs the code generated above into the general pattern:

```

fmod PRETTY-PRINT is
  protecting MONITOR-GENERATION .
  sort Monitor .
  op genMonitor : Formula -> Code .
  op makeMonitor : KConfiguration -> Code .

  var F : Formula . var B : Exp . var C : Code . vars N M : Nat .
  eq genMonitor(F) = makeMonitor(process(F)) .
  eq makeMonitor(k(exp(B)) code(C) nextBit(N))
    = bit[1 .. N] := false ;
    foreach new state s do {
      // "first update the bits in a consistent order"
      C ;
      // "then check whether the formula is violated"
      if !(B) then Error ;
      // "finally, update the state of the monitor"
      bit[1 .. N] := bit'[1 .. N]
    } .
endfm

```

Our implementation of the monitor synthesizer is now complete. To use it, one can ask Maude reduce terms of the form `genMonitor(F)`, where `F` is the formula that one wants to generate into a monitor. For example:

```

reduce genMonitor(
  !('a /\ !(0 'b /\ 'c S ('d /\ (! 'e S 'f)))
) .

```

For the formula above, Maude will give the expected answer, pretty printed as follows:

```

\|||||/
--- Welcome to Maude ---
/|||||
Maude 2.2 built: Mar 15 2006 16:37:22
Copyright 1997-2005 SRI International
Sat Jan 27 12:01:20 2007
Maude> in p
=====
fmod K
=====
fmod PREDICATE
=====

```

```

fmod SYNTAX
=====

fmod CODE
=====

fmod MONITOR-GENERATION
=====

fmod PRETTY-PRINT
=====

reduce in PRETTY-PRINT :
  genMonitor(! ('a /\ ! (0 'b /\ 'c S ('d /\ ! 'e S 'f)))) .
rewrites: 46 in -93406740ms cpu (1ms real) (~ rewrites/second)
result Code:
bit[1 .. 3] := false ;
foreach new state s do {
  // "first update the bits in a consistent order"
  bit'[1] := 'b(s) ;
  bit'[2] := 'f(s) \/ ! 'e(s) /\ bit[2] ;
  bit'[3] := 'd(s) /\ bit'[2] \/ 'c(s) /\ bit[3] ;
  // "then check whether the formula is violated"
  if 'a(s) /\ ! (bit[1] /\ bit'[3]) then
    Error ;
  // "finally, update the state of the monitor"
  bit[1 .. 3] := bit'[1 .. 3]
}

Maude>

```

The K Module One should upload the next module whenever one wants to use the K technique to define a language, logic or tool. Note that the module below has nothing to do with our particular logic under consideration in this paper; that is the reason for which we exiled it here.

```

fmod K is
  sorts KConfigurationItem KConfiguration .
  subsort KConfigurationItem < KConfiguration .
  op empty : -> KConfiguration .
  op __ : KConfiguration KConfiguration -> KConfiguration [assoc comm id: empty] .

  sorts KComputationItem KNeComputation KComputation .
  subsort KComputationItem < KNeComputation < KComputation .
  op nil : -> KComputation .
  op _->_ : KComputation KComputation -> KComputation [assoc id: nil] .
  op _->_ : KNeComputation KNeComputation -> KNeComputation [ditto] .

  sort KComputationList .
  subsort KComputation < KComputationList .
  op nil : -> KComputationList .
  op _,_ : KComputationList KComputationList -> KComputationList [assoc id: nil] .

```

```

sort KResult KResultList .
subsorts KResult < KResultList < KComputation .
op nil : -> KResultList .
op _,_ : KResultList KResultList -> KResultList [assoc id: nil] .

op [_] : KComputationList -> KComputationItem .
op [_] : KResultList -> KComputationItem .
op [_|_] : KComputationList KResultList -> KComputationItem .

var K : KNeComputation . var K1 : KComputationList .
var R : KResult . var R1 : KResultList .
eq [K,K1] = K -> [K1 | nil] .
eq R -> [K,K1 | R1] = K -> [K1 | R1,R] .
eq R -> [nil | R1] = [R1,R] .
endfm

```

To use K, after importing the module above, one should define one's own constructors for configuration items (sort `KConfigurationItem`), for computation items (sort `KComputationItem`), and for results (sort `KResult`). For our example, we defined all these at the beginning of the module `MONITOR-GENERATION`.

References

1. F. Chen, M. D'Amorim, and G. Roşu. Checking and correcting behaviors of Java programs at runtime with Java-MOP. In *RV'05*, volume 144(4) of *ENTCS*, 2005.
2. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude Manual. <http://maude.cs.uiuc.edu>.
3. K. Havelund and G. Roşu. Efficient monitoring of safety properties. *Software Tools and Technology Transfer*, 6(2):158–173, 2004. (also TACAS'02, LNCS 2280).
4. Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., New York, NY, USA, 1995.
5. G. Roşu. K: a rewrite-based framework for modular lang. design, semantics, analysis and implementation (V2). Technical Report UIUCDCS-R-2006-2802, 2006.